# NOP-Oriented Programming: Should we Care?

Pierre-Yves Péneau, Ludovic Claudepierre, Damien Hardy, Erven Rohou

# NOP-Oriented Programming: Should we Care?

Pierre-Yves Péneau
*Univ Rennes, Inria, CNRS, IRISA*
*first.last@inria.fr*

Ludovic Claudepierre
*Univ Rennes, Inria, CNRS, IRISA*
*first.last@inria.fr*

Damien Hardy
*Univ Rennes, Inria, CNRS, IRISA*
*first.last@irisa.fr*

Erven Rohou
*Univ Rennes, Inria, CNRS, IRISA*
*first.last@inria.fr*

***Abstract*—Many fault injection techniques have been proposed in the recent years to attack computing systems, as well as the corresponding countermeasures. Most of published attacks are limited to one or a few faults. We provide a theoretical analysis of instruction skip attacks to show how an attacker can modify an application behavior at run-time when thousands of instruction skips are possible. Our main result is that instruction skip is Turing-complete under our theoretical model while requiring the presence of only common instructions in the binary. As a consequence, we show that *current* software-based countermeasures are fragile. In addition, we release a modification of gem5 that implements a classical instruction skip fault model that we used for our experiments. We believe this kind of simulation tools are useful to help the community explore attacks and hardware and software countermeasures.**

***Index Terms*—fault injection, Turing-completeness, ISA-level simulation, instruction skip**

## 1. Introduction

Different fault injection techniques have been studied in the literature, such as causing clock or voltage glitches [2], using laser beams [33] or exploiting electromagnetic pulses [25] (EM). These techniques have been widely covered to inject a single or few faults to hijack control flow or change a register value.

Conversely, protection mechanisms, at the software level [3], [5], [26], have been proposed to protect against few faults injection. At the hardware level, protections [1], [21], [38] have been proposed as well, but they may not always be present for on-the-shelf hardware.

Thanks to all the studies, a better understanding has been reached and lots of attacks have been proposed [4], [11], [20], [22], [24], [29].

Fault injection is used to achieve two objectives: corrupt instructions or corrupt data. These are not exclusive and an attacker might be able to do both. This paper focuses on the first objective: instruction corruption. More specifically, we focus on instruction skip, *i.e.*, the possibility to skip the entire execution of an instruction at runtime. This is what we call a NOP-oriented programming.

Recent work shows that it is possible to inject multiple faults while keeping the target program in a non-crashed state [9]. In this paper, we propose a theoretical analysis where a large amount of precise and reliable fault injection is possible and explore the consequences of this assumption. This includes the ability to impact any given instruction (determined by the exact clock cycle at which the fault shall occur), and to induce the fault at any CPU frequency. In particular, we consider bursts of faults where hundreds of cycles can be impacted.

In such a situation, new questions arise: *i)* What would be the possibilities for attackers with multiple fault injection? *ii)* Are existing protections still effective? *iii)* If not, how to protect our systems against these attacks?

To study these questions, this paper makes the following contributions:

- a theoretical analysis of the security threats with multiple instruction skip which shows the Turing-completeness of our theoretical model;
- a modified version of gem5 to verify the previous analysis and allows exploring new possibilities through simulation.

This paper is structured as follows. Section 2 details previous related work. Section 3 proposes a theoretical analysis of instruction skip, Section 4 presents a simulator which implements an instruction skip fault model. Experimental results from simulations are presented in Section 5. We discuss our study in Section 6 and conclude in Section 7.

## 2. Related work

In this paper, we consider a fault model where an attacker is able to entirely skip a specific instruction or a set of specific instructions [19], [20], [22]. Previous work has demonstrated the feasibility of skipping a single instruction with different injection types such as electromagnetic [25] (EM), laser [33] or clock [2] and voltage glitches [37]. Skipping an instruction can be achieved by several means: *i)* alter the bits of the opcode and substitute the executed instruction by another one without side-effect (considered as a nop) [25]; *ii)* modify the Program Counter (PC) [35]; *iii)* target the instruction buffer [31].

All of these works target embedded systems. Recent publications show an on-going trend to put the effort on high-performance system-on-chips (SoC). Majeric *et al.* [24] demonstrate the feasibility of EM injection on a

Cortex-A9 co-processor dedicated to AES. They identified the most sensitive area to injection and are able to produce different faults to obtain information from the co-processor. Proy *et al.* [29] tackle SoC security with a more theoretical approach. They study the effect of EM pulse injection on the Instruction Set Architecture (ISA) of the ARM Cortex-A9 and propose a fault model.

Different countermeasures have been proposed to detect, avoid and correct these attacks. They are applied at software [6] or hardware [38] level, or both. One of the most used mechanism is redundancy [3], implemented at different levels: function or instruction. In the former, a block of instructions (*i.e.,* a complete function) is executed twice and the results are compared. If the results are the same, the execution is considered as safe. In the latter, granularity is at the instruction level. Each instruction is executed twice, or $n$ times [27], [30]. It has been completed by the concept of idempotent instruction developed by Moro *et al.* [26] and completed by Barry *et al.* [5]. An idempotent instruction is an instruction that does not modify its input registers. Hence, this instruction can be repeated $n$-times without side effect. These proposals prevent an attacker from injecting a fault at specific instruction since they have to target multiple instruction at the same time. However, it has been proven not secure and the source of information leakage for side-channel attack [13]. Another possibility is to detect timing faults [1], [21], [39] or clock/power glitches [38] by adding new hardware.

However, despite a large effort from the community, it has been shown that the injection of multiple faults at a very precise point during the execution with the desirable effect still requires a very expensive setup [4] or has a low rate of success [9]. Moreover, keeping the processor on a non-faulty state is a challenging task [22], [36]. Thus, this has long been considered as a non-realistic fault model. Nonetheless, we believe this could be achieved in a near future with lower requirements and we advocate for a theoretical analysis of the possibilities of these attacks. This work proposes such an analysis. We consider a fault model where the attacker can skip a high number of instructions without side-effects and explore the theoretical possibilities of such a feature. Moreover, we explore its feasibility through the use of a the gem5 [8] simulator.

# 3. Potential of Many-Fault Attacks

In this section, we propose a theoretical analysis of the potential of many-fault attacks. All actual practical limitations such as the number of faults that can be injected, their injection at the correct location, or their effects like replay of instructions [31] or data corruption are ignored to focus on the possibilities given to the attacker.

Assumptions concerning the fault model for the theoretical analysis are provided in 3.1 followed by the program alterations that can be derived from this threat model in 3.2. A proof that NOP-Oriented Programming is Turing-complete under these assumptions is then detailed in 3.3. Finally, some uses cases are discussed in 3.4.

## 3.1. Assumptions

For the theoretical analysis of multiple faults injection, the following is assumed. A fault injection at run-time cancels the execution of any possible instruction, practically replacing it by a nop. An infinite number of nop injections is assumed to be possible during program execution.

Concerning the application subject to such an attack, it is assumed that common assembly instructions (*i.e.* `load`, `store`, `move`, `add`, `sub`, `mult`) are present in the binary, and the application is bug free. In particular, we do not require the presence of vulnerabilities, such as buffer overflow or fault activated backdoor.

## 3.2. Application behavior modifications

In this part, we describe some program modifications that can be achieved by multiple nop injection. All the presented modifications were tested on a STM32F100RB board running the ARM instruction set. For the sake of simplicity, we directly modified the source code of applications at assembly level and replaced selected instructions by nops. This is more restrictive (*i.e.* less powerful) than an attack at run-time: in our setup all instances of such instructions become nops, while a real attacker would have the liberty to cancel only selected instances (such as selected iterations of a loop), thus generating even more diverse instruction sequences.

***Modification 1.*** Hijacking the Control Flow Graph
This modification for single nop injection has been previously studied by Bukasa *et al.* [11] to perform control flow hijacking or fault activated backdoor.
With multiple nop injection, any instruction that can be reached from a program point $p$ can be executed immediately after $p$. The reachable instructions are all the instructions that have an address higher than or equal to the reachable instruction with the smallest address. This can be done thanks to the fact that:

- each instruction can be executed or not;
- edges can be added to the control flow graph between each jump instruction (branch, call, return) and the instruction immediately following the jump in the binary, as illustrated in Figure 1. In other words, all control-transfer instructions become conditional (including function returns).

Notice that, in some situations depending on the layout and/or the presence of special functions, like handlers that implement a software reset, any instruction in the binary can be reached from nearly any program point. Furthermore, faulting specific instructions like `push` and `pop` make it possible to control the stack pointer. Finally, pure software protections can be simply skipped, including hardened code [28] or Return-Oriented Programming (ROP) [10], [32] protections.

***Modification 2.*** Altering the number of loop iterations
Each loop trip count can be altered from 0 to infinite. This modification is a direct extension of modification 1. Reducing the number of iterations can be done by two ways:

- replacing the entire loop body by nop instructions;
- replacing the (un)conditional branch instruction by a nop.

On the opposite side, the number of iterations can be increased by:

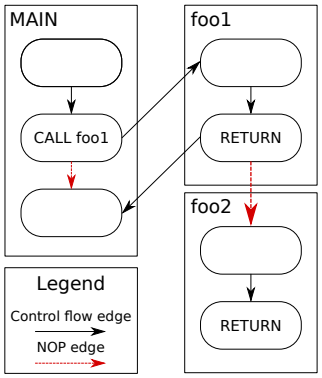



Figure 1. Example of a control flow graph with edges added with the NOP injections

- replacing the compare instruction by a nop. In that case, the condition flags may have to be set before the conditional branch to ensure that the branch is taken;
  or
- replacing the instructions that control the loop condition by nops. Example:

```
label:
  ...
  sub r, #1 // nop instead
  cmp r, #0
  bne label // branch if not equal
```

  In this case, register $r$ cannot be used by the nop injection attack, except if it is possible to ensure that the condition to leave the loop returns false. In the previous example, to use $r$, it is mandatory to ensure that $r$ is always different from 0 to stay in the loop.

Note that, bare metal applications for embedded devices are generally composed of at least a main infinite loop (after initialization).

***Modification 3.*** Write any possible values in a register
To be able to set any value in a register $r$, the binary must have a set of instructions that modify $r$ in a way that any possible values can be computed by executing these instructions multiple times, and it is possible to control the number of times they are executed (inside a loop subject to modification 2 for instance). The other instructions that modify $r$ are replaced by nops.
Example:

```
loop:
  ...
  mov r, #0 // executed only once, then skipped
  add r, #1
  b   loop  // jump
```

Note that, a load instruction to $r$ from a memory location that can be affected by an input function provides the same effect.

***Modification 4.*** Write any possible values in a set of registers $R$
This modification is an extension of the previous modification 3:

- modification 3 can be applied on each register $r \in R$;
  or
- after setting register $r \in R$ to a targeted value, the program has reachable instructions to copy the content of register $r$ to $r' \in R$.
  Example 1:

```
mov r', r
```

  Example 2:

```
store r,@
load  r',@
```

***Modification 5.*** Load/Store any possible values in/from register $r$
For this modification, we assume that the binary has a reachable instruction to load/store a value in register $r$ from/to the address contained in register $r'$. The memory location must have readable (respectively writable) permission for the load (respectively store).
For the load case, modification 3 is needed to set $r'$ to a targeted address. For the store, modification 4 is needed to set $r'$ to a targeted address and $r$ to a targeted value.
Note that, for the store instruction, this modification allows to change the stack content and thus allows Return-Oriented Programming attack. Furthermore, new binary instructions can be generated and stored in RAM to be executed later. In some cases, it can be stored in Flash or NVRAM to alter permanently the original application.

***Modification 6.*** Jump to any address with execute permission
This last modification allows to jump directly to any addresses with execute permission thanks to modification 3 that can set register $r$ to a targeted address and the presence of a reachable instruction to jump to it.
Example 1:

```
  blx r
```

Example 2:

```
  push r
  pop  pc
```

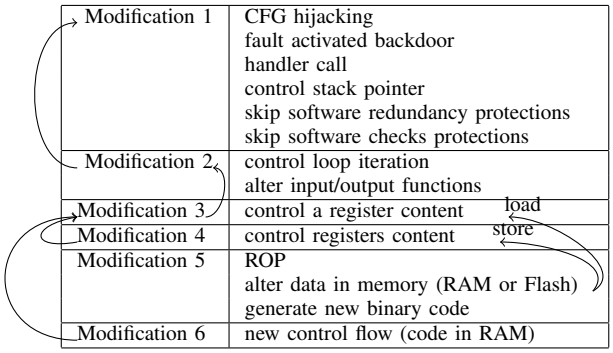
Note that, combined with modification 5, it is possible in some cases to generate binary code in RAM and jump to it.

To summarize all these modifications, Table 1 provides the direct dependencies between the modifications and shows some examples of attacks when the modifications are available. It can be noticed that modifications 1, 3 and 4 are keys to perform others modifications.

### 3.3. NOP-Oriented Programming is Turing-complete

As generally assumed to prove that a language is Turing-complete, we only focus on the computation part, and ignore the output function. It can be noticed that the input function is not mandatory in our case.

Based on the modifications of the previous section, it is obvious that multiple nop injection is Turing-complete in some specific cases with the Return-Oriented Programming equivalence (modification 5) or the possibility to




| | |
|---|---|
| Modification 1 | CFG hijacking<br>fault activated backdoor<br>handler call<br>control stack pointer<br>skip software redundancy protections<br>skip software checks protections |
| Modification 2 | control loop iteration<br>alter input/output functions |
| Modification 3 | control a register content |
| Modification 4 | control registers content |
| Modification 5 | ROP<br>alter data in memory (RAM or Flash)<br>generate new binary code |
| Modification 6 | new control flow (code in RAM) |

TABLE 1. EXAMPLES OF POSSIBILITIES GIVEN TO THE ATTACKER BASED ON THE MODIFICATIONS. AN EDGE INDICATES THAT MODIFICATION B DEPENDS ON MODIFICATION A (B→A)

generate new binary instructions in RAM (modification 5) and jump to it (modification 6).

This intuition can be generalized in case the following modifications are available:

- modification 4 for few (at least three[1]) registers
- modification 5 for these registers
- the binary has instructions that can be used to perform NAND operations on these registers
  - `and` and `not`
  - multiplication and subtraction
  - ...
- all of these are available in a loop that can be possibly created by nop injection and subject to modification 2.

**Proof.** With these modifications it is possible to:

- store 0/1 values in any memory location an infinite number of time;
- load values from any possible memory location an infinite number of time.

Furthermore, the NAND operation is known for its property of functional completeness. In particular, AND, OR, XOR and NOT operations can be computed only with NAND operations. So the following boolean expression $(\neg p \wedge q) \vee (q \oplus r)$, that correspond to the Rule 110 cellular automaton which is known to be Turing-complete [14], [15], can be computed. □

Another possibility to prove that it is Turing-complete would be to use the method proposed in [18] based on move instructions. This approach will require `move`, `load`, `store` and `jump` instructions and the same modifications but will avoid the need of NAND operations.

### 3.4. More realistic use-cases

The previous section focuses on purely theoretical aspects that require potentially a huge amount of fault injection to create an attack. In the following, we present some use-cases that require a reasonable amount of nops, and that have been tested on real hardware (STM32F100RB and STM32F3030RE) by inserting nops at the assembly level. The applications are compiled with the default flags provided by STM32CubeMX. We rely on the hardware abstraction layer (HAL) provided by the STM32 API, which is widely used to program such systems. Our analysis applies to all applications based on this API or similar APIs with the same functionalities.

1. The actual number of required registers depends on the ISA instructions and the available instructions in the binary to perform load, store and NAND operations.

**Dump part of the firmware or data.** For this use case, we modify only the function `HAL_UART_Transmit()` to be able to dump pieces of code (in binary) or some data. This function has four parameters: the address to the UART handler, the address of the information to transmit, the size in bytes of the information, and a timeout. They are all passed by registers, in our case `r0` to `r3`.

By changing the parameters (*i.e.* performing a nop when the parameter is set) before the call, the address of the information or its size can be altered to transmit more information and from a different location in the application binary. Furthermore, inside the function `HAL_UART_Transmit()`, a loop is used to countdown the number of transmitted bytes. By applying modification 2, the number of transmitted bytes can be increased.

**Dump a key.** Let us assume that we have an application that transmits encrypted information as follows:

```
1 Encrypt(key, plaintext, ciphertext)
2 HAL_UART_Transmit(uart, ciphertext, size, timeout)
```

which corresponds to the following code at the pseudo assembly level:

```
1 mov r0, @key
2 mov r1, @plaintext
3 mov r2, @ciphertext
4 call Encrypt
5 mov r0, @uart
6 mov r1, @ciphertext
7 mov r2, size
8 mov r3, timout
9 call HAL_UART_Transmit
```

If inside the function `Encrypt()` (or any reachable instructions from this function) there is a way to transfer the content of register `r0` to `r1` (`mov r1, r0` for instance), we can nop all the instructions that affect the content of `r0` before the transfer, all the instructions that affect `r1` after the transfer including line 6. In that situation, the key address will be transmitted instead of the ciphertext address to the `HAL_UART_Transmit()` function. It can be noticed that if the register used for the key address in function `Encrypt()` is the same as the register used for ciphertext address in function `HAL_UART_Transmit()` only two nops are needed: one for line 4 and the other one for line 6.

**Load custom code in RAM and execute it.** For the last use-case, we attack the function `HAL_UART_Receive()` and we assume that the provided input can be controlled by the attacker. The signature of function `HAL_UART_Receive()` is very similar to `HAL_UART_Transmit()` except that the second parameter is the address where to store the received data.

To be able to load enough binary code in RAM, the loop inside `HAL_UART_Receive()` that counts down the number of received bytes is altered by applying modification 2.

To test that attack, we loaded two handwritten binaries: $(i)$ one that performs a classical *Hello World!* that is able to call functions in the original binary and $(ii)$ a loader that can be used once activated to load new binaries in RAM and then execute it like the *Hello World!*.

For handwritten binary code to be able to execute in RAM, all addresses have to be hard coded and thus it strongly depends on the application layout. Especially in our case for the addresses of functions `HAL_UART_Transmit()` and `HAL_UART_Receive()` as well as the address of the UART descriptor. Note that, to write data (*i.e.* the binary code) in RAM, addresses are not protected on our STM32 boards. This allows a certain degree of liberty for its location. However, we have to be careful not overwrite useful data, in our case the UART descriptor.

For testing purpose, we use a hard-coded instruction with the correct address to jump to the correct location in RAM. This can be achieved by a `pop {pc}` instruction or a `blx r` instruction.

In case the board like the STM32F3030RE has a memory protection unit to forbid execution from addresses in RAM, there are several ways to deactivate it, the simplest one is to inject nops during its activation.

To go one step further, and to take into account more realistic fault injection models and help defining protections against such kind of attacks, we detail a modified version of gem5 in the next section.

## 4. Simulation setup

We propose a modification of the gem5 simulator [8] to simulate attacks, explore its possibilities and propose countermeasures. This modified version of gem5 is available online[2]. gem5 is a widely used simulator in the micro-architecture community, supported by large company like Google, ARM or AMD and known for its accuracy [12]. gem5 is open-source and is strongly object oriented, which facilitates any modification of the code.

### 4.1. Fault model

We consider a fault model where an instruction is not strictly replaced by a nop. Instead, we repeat the instruction before the one we want to skip. This model is similar to what Riviere *et al.* [31] experienced. The main difference here is the number of repeated instructions. Riviere *et al.* repeat the last four instructions. In their setup, the instruction buffer has a capacity of 128 bits and 32 bits are used per instruction. Thus, it keeps in memory the last four instructions.

Considering the code in Listing 1, one expects a result of 6 in `r0` at the end of the sequence. If we attack instruction on line 3, the result in `r0` is 3, since instruction one line 2 is executed twice.

```
1 mov r0,      #0
2 add r0, r0, #1
3 add r0, r0, #4    // skip here
4 add r0, r0, #1
```

Listing 1. Addition in ARM assembler, attack is on instruction 3

2. https://gitlab.inria.fr/gem5-nop/gem5

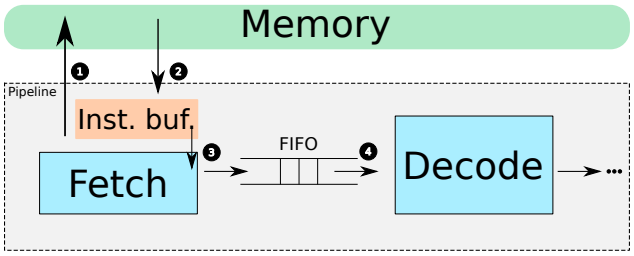



Figure 2. Functional view of the fetch and decode stages

In terms of micro-architecture, we target *fetch requests*. Figure 2 depicts a normal situation in the pipeline. The fetch stage initiates a *fetch request* (step 1) to the memory if the next instruction is not in the instruction buffer. Then, the instruction is loaded from the memory to the instruction buffer (step 2) and copied to the FIFO between the fetch and decode stages (step 3). Then, the decode stage reads the FIFO and executes the instruction.

In this work, we consider an instruction buffer of size 1. Hence, at each cycle, the core needs to fetch the instruction from the memory to the instruction buffer. At cycle $T$, the instruction buffer contains instruction $I$. By preventing step 1 at cycle $T+1$, we avoid step 2 which erases the content of the instruction buffer. This means that the instruction buffer stills contains instruction $I$ at $T+1$, instead of instruction $I+1$. As a consequence, instruction $I$ is executed at cycle $T+1$. However, we do increment the PC to fetch instruction $I+2$ at cycle $T+2$.

It is important to mention that attacks on instruction buffer only works with fixed-size instruction set, *e.g.* RISC architectures. Regarding side-effects of repeated instructions, they are not important for the attacker if these side-effects do not alter the ongoing attack.

### 4.2. Usage and limitations

We modified the gem5 source code and created the `NopsSimpleCPU` based on the `TimingSimpleCPU`. We added a command-line option that takes a file in argument: `--nops-file`. This file contains a list of physical addresses with specific actions to do: skip or execute. To do so, the `fetch()` method is overridden. Before sending a request for instruction $I$ to the memory, the CPU checks if there is an action related to this address, and acts accordingly. In case of a skip, no request is sent, but the CPU state is advanced as usual, especially the program counter.

Instructions, specified by physical addresses in the configuration file, can be:

a) skipped each time
b) skipped a given number of time
c) executed a given number of time
d) a combination of *b* and *c*

These options, specified in the input file, are set using this template: `addr s:X e:Y .. .` Several examples are given in Listing 2. The first line of the configuration file defines the address where the attack begins. No instruction is skipped until this address is executed $N$ times, $N$ specified on the same line than the address.

```
10528 1              // beginning of the attack
10654                // case a
10658 s:10           // case b
1065c e:5            // case c
10660 s:1 e:10 s:2   // case d
```

Listing 2. Example of how to use the nop feature

As shown in Listing 2, the simulator starts to skip instructions when address `0x10528` is executed once. Line 5 gives an example of a combination of skip and execute. The instruction at address `0x10660` is skipped once, executed 10 times, skipped 2 times then always executed.

In the future, we plan to extend this fault model to a more precise CPU model provided in gem5. This would have two major benefits: *a)* a configurable size for the instruction buffer to study the impact of more replayed instructions; *b)* a more detailed pipeline model to assess effects of the replay in this part of the core. Providing a more realistic simulation tool would allow researchers to study and propose countermeasures against an attacker with a capability of large instruction skip.

## 5. Experimental results

In this section, we propose to apply two use-cases mentioned in Section 3.4: *i)* a dump of the key used in an encryption process, *e.g.* AES algorithm, and *ii)* generating custom data in memory. The first use-case is derived into five different attacks.

Note that this work is independent of the encryption algorithm. We do not demonstrate an attack on AES specifically, it has been well covered elsewhere [4], [7], [23], [34]. Our proposal is generic and can be applied to any sort of code, depending on the presence of the basic set of assembly instructions required (see Section 3). In order to illustrate this generic process, the AES encryption function is named `Encrypt()` in the rest of this paper.

In this study, we consider a white box model where the attacker has access to the source code to insert a trigger to synchronize the attack. Synchronization issues or source code access are out of the scope in this work. In gem5, the trigger is the address defined on the first line of the configuration file. Source code in C are compiled with the following flags: `-static -O0 -fno-builtin`. The `static` flag is due to gem5 which cannot execute non-static programs without a Linux layer. Disabling optimisation through `-O0` is a common practice in terms of security. Since security is not defined by the language standard, compiler optimizations will attempt to eliminate redundancies and tend to remove security checks [17].

### 5.1. Dump of a secret key

We propose five different versions of this use-case. The first two are easy to patch by the compiler while the three other are more sophisticated.

**Naive implementation.** This attack is based on modification 3 defined in Section 3.2, *i.e.,* the possibility to write any value in a register.

Consider the C code and its equivalent in assembler, respectively depicted in Listings 3 and 4. The address of `key` is loaded into `r1` before calling `Encrypt()`. After the call, `printf()` uses `r1` as the register which

```
Encrypt(cipher, key, plaintext);
printf("%s\n", cipher);
```

Listing 3. Naive `main()` function in C

```
10594: ldr  r2, [fp, #-8]
10598: ldr  r1, [fp, #-16]
1059c: ldr  r0, [fp, #-12]
105a0: bl   105d0 <Encrypt>
105a4: ldr  r1, [fp, #-12]
105a8: ldr  r0, [pc, #28]
105ac: bl   17cc4 <_IO_printf>
```

Listing 4. Attack performed on the naive `main()` function

contains the address of the buffer to print. By skipping instructions at addresses `0x105a0` and `0x105a4`, we are able to print the key in the standard output instead of the cipher. Note that skipping those two instructions means repeating the instruction at address `0x1059c` two times more (three times in total). However, this repetition has no side-effect on the program execution. Hence, only two nop are required to retrieve the key.

**Use the program layout.** This attack is based on modification 1 and modification 3 defined in Section 3.2, respectively the CFG hijacking and the possibility to write any value in a register. It remains an easy-to-fix attack in the compiler.

In this use-case, we consider the same functions as presented in Listing 3, but the `key` argument is the third parameter. Hence, its address is stored in register `r2` before calling `Encrypt()`. To retrieve the key, an attacker needs to move the content of `r2` into `r1`. One could see this change as a simple countermeasure regarding the previous attack. Moreover, the program layout is as described by Listing 5, *e.g.,* the `Encrypt()` function is located ahead of the `main()` function.

A partial dump of the `Encrypt()` functions is shown on Listing 5. One observes that `r2`, the address of the key, is stored on the stack at `sp+4`. This store is almost immediately followed by a load of this address into register `r1`. To print the key, we do a contiguous burst of nop, from the `Encrypt()` function to the `main()` function. In our setup, this burst goes from `0x10544` to `0x11494` included. This represents 981 nop, or, 981 repetition of load instruction at `0x10540`.

Note that we choose to do a burst for the sake of simplicity. For a less invasive attack, one has to skip only the instructions that modify register `r1` and the `Encrypt()` return. This strategy of burst is re-used for the following use-cases.

**Use an handler function.** The attack relies on the existence of at least one handler function like `abort()` or `exit()` located at the beginning of the program, or at least before the `main()` function. It is based on modification 1 and modification 3. Compared to a real use-case on the STM32 boad, it is equivalent to inject a fault that generates a call to `HardFault_Handler()`, which resets the board and invokes the `main()` function. Moreover, we consider a layout described in Listing 6, *i.e.*, where the `main()` function comes before `Encrypt()` This layout ensure minimal protection regarding of the previous use-case. We assume the order of the arguments as in the previous example: cipher, plaintext and key. The

```
<Encrypt>:
 10528: push    {lr}
 1052c: sub     sp, sp, #252
 10530: str     r0, [sp, #12]
 10534: str     r1, [sp, #8]
 10538: str     r2, [sp, #4]    // Store r2
 1053c: add     r3, sp, #228
 10540: ldr     r1, [sp, #4]    // Load into r1
 10544: mov     r0, r3
        ...
        ...
<main>:                                  nop burst
        ...
 11490: bl      10528 <Encrypt>
 11494: ldr     r1, [sp, #8]
 11498: ldr     r0, [pc, #28]
 1149c: bl      17c44 <_IO_printf>
```

Listing 5. `Encrypt()` instructions used to move `r2` to `r1`. Burst starts at `0x10544` and stops at `0x11494`

```
<abort>:                                 4: nop
        ...                              burst
<main>:
        ...
10594: ldr r2, [fp, #-16]
10598: ldr r1, [fp, #-8]
1059c: ldr r0, [fp, #-12]
105a0: bl 105d0 <Encrypt>
105a4: ldr r1, [fp, #-12]
105a8: ldr r0, [pc, #28]                 3: jump
105ac: bl 17cc4 <_IO_printf>
        ...
<Encrypt>:
105d0: push {fp, lr}
105d4: add  fp, sp, #4
105d8: sub  sp, sp, #248
105dc: str  r0, [fp, #-240]
105e0: str  r1, [fp, #-244]
105e4: str  r2, [fp, #-248]              1: inst.
105e8: sub  r3, fp, #24                  to reach
105ec: ldr  r1, [fp, #-248]
105f0: mov  r0, r3
        ...                      2: nop
<__assert_fail_base>:            burst
        ...
11ef4: bl 10170 <abort>
```

Listing 6. Control Flow Hijacking to retrieve the secret key. Instructions in `main()` are at least executed once without nop to reach step 1 in `Encrypt()`

objective is identical: move the address of the key from `r2` to `r1`, then find a way to the `printf()` call in the `main()` function.

To reach this goal, we use the same sequence of instructions in `Encrypt()`, as in the previous example to move the key address from register `r2` to `r1` (step 1). Note that addresses are different in Listing 6 and Listing 5 due to a different layout. Then, we initiate a burst of 1601 nop to reach address `0x11ef4` (step 2). As shown in Listing 6, this address corresponds to a branch to the `abort()` function (step 3). In terms of physical addresses, this function is located ahead of the `main()` function. After, we initiate a second burst of nop from this point to reach the call to the `printf()` function in the `main()` (step 4). This burst is composed of 269 nop.

**Generic attack.** This example illustrates the usage of modification 1 and modification 4, *i.e.*, CFG hijacking, the ability to write any value in a set of registers and no other assumptions. Listing 7 shows the relevant part of the code and the scenario of the attack.

From `main()`, the application jumps to `Encrypt()` with the address of the secret key stored in `r2`. This address is then stored at `sp+4` (`0x105dc`) and loaded

```
<main>:
        ...
10590: ldr     r2, [sp, #4]
10594: ldr     r1, [sp, #12]
10598: ldr     r0, [sp, #8]
1059c: bl      105cc <Encrypt>
105a0: ldr     r1, [sp, #8]
105a4: ldr     r0, [pc, #28]
105a8: bl      17c44 <_IO_printf>
        ...                                  3: jump
<Encrypt>:
105cc: push    {lr}
105d0: sub     sp, sp, #252
105d4: str     r0, [sp, #12]
105d8: str     r1, [sp, #8]
105dc: str     r2, [sp, #4]   // key addr
105e0: add     r3, sp, #228
105e4: ldr     r1, [sp, #4]
105e8: mov     r0, r3                        1: nop
        ...                                  burst
<F>:
10ef4: sub     sp, sp, #24
10ef8: str     r0, [sp, #12]
10efc: str     r1, [sp, #8]
10f00: str     r2, [sp, #4]                  2: nop
        ...                                  burst
10f94: nop
10f98: add     sp, sp, #24
10f9c: bx      lr
```

Listing 7. Control Flow Hijacking and load/store on registers to retrieve the secret key. Instructions are normally executed from `main()` until `0x105e8` where the first burst begins

into `r1` (`0x105e4`). After the execution of this instruction, we initiate a first burst of 581 nop to execute the instruction at `0x10efc`: `str r1, [sp #8]`. At this point, we have successfully transferred the address of the key from `r2` to `r1`, and then stored this address at `sp+8`.

We then initiate a second burst of 39 nop to reach the last instruction of the `F()` function: `bx lr`. The execution of this instruction triggers a return to the main, which continues its execution normally. Arguments for `printf()` are prepared and the buffer address to print is loaded from `sp+8`, which now contains the address of the key.

Note that purpose of the `F()` function is not important. The objective is to reach the following instructions: `str r1, [sp, #8]` and `bx lr`. These instructions are commonly used by the compiler. Hence, our attack does not rely on this particular implementation.

**Another generic attack.** In the previous example, we retrieve the key by calling the `Encrypt()` function and use an appropriate instruction. In this case, we show that it is still possible to retrieve the key without starting the encryption process. The attack is described in Listing 8.

First, we skip the call to `Encrypt()` in the main to reach `printf()`. At this point, `r0` contains the address of the string format required by `printf()`. One observes on line 8 that loading the right address into `r0` is pc-relative. Therefore, it is not possible to skip this call to `printf()`. Moreover, we have to avoid any modification of `r0` from now until the end of the attack.

The next objective is to move the address of the key from `r2` to `r1`. A first burst of 18 nop is initiated to reach instruction `bx lr` at the end of `printf()`. The execution of this instruction modifies the program counter to `0x105ac`, *i.e.,* the next instruction after the branch. Then, we start a second burst of 22 nop from `0x105b0` to `0x10604` and execute the instruction `mov r1, r2`.

From this point, register `r1` contains the memory

```
<main>:
        ...
10590: ldr     r2, [sp, #4]
10594: ldr     r1, [sp, #12]
10598: ldr     r0, [sp, #8]
1059c: bl      105cc <Encrypt>
105a0: ldr     r1, [sp, #8]
105a4: ldr     r0, [pc, #28]
105a8: bl      17c44 <_IO_printf>
105ac: mov     r3, #0
105b0: mov     r0, r3
105b4: add     sp, sp, #20
105b8: pop     {pc}
        ...
<Encrypt>
        ...
10600: add     r2, sp, #228
10604: add     r3, sp, #20
10608: mov     r1, r2
1060c: mov     r0, r3
        ...
        ...
<_IO_printf>:
17c44: push    {r0, r1, r2, r3}
17c48: push    {r4, lr}
17c4c: ldr     r4, [pc, #76]
17c50: sub     sp, sp, #8
17c54: ldr     r1, [r4]
        ...
17c90: pop     {r4, lr}
17c94: add     sp, sp, #16
17c98: bx      lr
```

2: jump

3: nop burst

4: nop burst

1: nop burst

Listing 8. Control Flow Hijacking and move on registers to retrieve the secret key.

address of the key. The last step is to reach the `printf()` function by using a large burst of 7566 nop. When the program counter is at `0x17c44`, we stop the burst and retrieve the key.

## 5.2. Generate custom data in memory

This use-case illustrates the third example presented in Section 3.4, *i.e.,* the code injection and execution. However, this illustration slightly differs from the example: we do not load code but generate directly data in memory. This is mainly due to gem5 that prevents user to execute code from memory[3]. Still, the idea is identical and has been successfully tested on the STM32F3030RE and STM32F100RB boards.

The base code remains the same as in the previous use-case: an encryption and a print of the ciphertext. Here, we take control of the memory buffer used by `printf()` to display a classic *Hello World!*. This sequence of characters is generated from scratch by using instructions in the code like `add`, `load` and `store`. To achieve this, we apply modifications 1, 2, 3, 4 and 5 presented in Table 1.

The attack is presented in two phases. The first one explains step by step the spirit of the attack. The second phase is the implementation.

**Attack step by step.** The following steps present how the attack is constructed. All these steps are mandatory to generate the data in memory.

1) take control of the number of iterations of a loop
2) inside this loop, take control of two registers
3) iterate over the loop to generate the memory content in the first register and the memory address where the generated data will be stored in the second register. In our example it corresponds to the address of the ciphertext.
4) exit the loop and find a `store` instruction that uses the two controlled registers and execute it.
5) return to the controlled loop and repeat the process for other characters. When it is done, call the `printf()` function with the correct parameters.

3. Changing this behavior is part of our ongoing work in gem5.

```
<Encrypt>:
        ...
1061c: mov  r3, #1          // Init. r3 to 1
        ...
10628: ldr  r3, [pc, #264] // Skipped 1st time
        ...
10634: ldr  r3, [sp, #244] // Get r3 from stack
10638: cmp  r3, r2          // Reset comp. flags (once)
        ...
1067c: ldr  r2, [sp, #244] // Get ASCII in r2
        ...
106a0: add  r3, r3, #1      // Add +1 to r3
106a4: str  r3, [sp, #244] // Store r3 in stack
        ...
106ac: cmp  r3, #9          // Exec at last iteration
106b0: ble  10628 <Encrypt+0x68>
        ...
106e4: bl   1073c <G>
        ...
10734: pop  {pc}

<G>:                        // Function G is only used to
1073c: sub  sp, sp, #8      // to write the generated
        ...                 // ASCII code in r2 at
1075c: strb r2, [r3]        // the generated address in r3
        ...
10968: add  sp, sp, #8
1096c: bx   lr
```

1: Main loop to generate address and ASCII

Listing 9. Main loop hijacked to generate memory addresses and characters to print *Hello World!*

**Implementation.** The code is shown on Listing 9. For reading-ease, we only show executed instructions. The full and detailed attack is provided with the source of our modified version of gem5. Below, we describe how each step of the previous section is implemented.

***Step 1:*** the loop goes from `0x10628` to `0x106b0`. The `cmp` instruction before the branch is never executed, except at the last iteration to exit the loop.

***Step 2:*** we control register `r3` through the `add` instruction at `0x106a0`. This instruction is executed at each iteration of the loop. We also control `r2` with instruction `load` at `0x1067c`.

***Step 3:*** iterate over the loop until register `r3` contains the memory address where data will be stored. Moreover, when `r3` contains the appropriate ASCII value, *e.g.*, 72 for 'H', execute the load instruction at `0x1067c`.

***Step 4:*** we execute the `cmp` instruction at `0x106ac` to set comparison flags to their appropriate values. Then, the following branch is not taken and the loop is over. We skip few instructions at the end of the loop to reach a branch to function `G()`. In `G()`, all instructions are skipped except *(i)* the first and last one that manipulate the stack and *ii)* the store instruction `strb r2, [r3]` which uses our two controlled registers. We execute the store, exit function `G()`, return to `Encrypt()` and then return to `main()`.

***Step 5:*** to loop into the `Encrypt()` function, we use an infinite loop in the main. As mentioned before, bare metal applications for embedded devices are generally

composed of at least a main infinite loop. Note that reaching the `main()` function could be achieved by using the handler technique used in Section 5.1. We repeat this process 12 times to write all the characters from the *Hello World!* string. To keep the control of the loop in `Encrypt()` between two calls, we reset the comparison flags by executing once the compare instruction at `0x10638`. Calls to `printf()` in the main function are always skipped, except when the *Hello World!* string has been fully generated in memory.

As previously said, this method works to write a single byte in memory. This is specific to instructions available in `Encrypt()`. A store on 4 bytes instead of 1 would have simplified the process. More optimized versions are possible but not presented in this work.

## 6. Discussion

In this paper, we consider a scenario where an attacker is able to inject a large number of faults in a precise way.

In Section 3, we detailed some modifications of the application behavior that an attacker can combine to perform attacks. Our main result is that it is Turing-complete under our theoretical model while requiring only very common instructions that are present in most binaries. In other words, it means that an attacker will only have to select among available instructions to execute what they want to execute.

Our theoretical fault model proposed in Section 3 does not reflect the one experienced by Riviere *et al.* [31]. In Section 4, we present an instruction skip fault model implemented in the `NopsSimpleCPU` model of the gem5 simulator. With this model, we are able to skip a single instruction by repeating the previous one. Using gem5, we show in Section 5 that it is possible to perform similar attacks as Riviere *et al.* and even more sophisticated ones. We believe that without appropriate protections, most of the theoretical attacks are implementable on realistic models, as long as enough injections are available.

Concerning software protections, based on modification 1, it is possible to skip theoretically any instruction, especially those that are added to check the application behavior. Thus, most if not all, existing protection techniques can be in theory skipped. In practice, the possibilities to skip specific instructions may be affected by countermeasure that randomize timing [16] or physical limitation (*e.g.,* a delay between two electromagnetic pulses). This would limit the range of attacks that can realistically be performed. Countermeasures are part of our ongoing work that we plan to investigate on our modified version of gem5. We also plan to build an experimental setup for large instruction skip to experimentally assess the viability of the theoretical model presented in this paper.

Concerning existing hardware protections, they are not always present for on-the-shelf hardware. Even though there exist hardware protections against instruction skip, there is still room to provide new low-cost hardware protections, possibly relying on pieces of software. We plan to upgrade our modified version of gem5 to provide a framework with more detailed micro-architecture to investigate new hardware protection mechanisms as well as combined hardware and software protections.

## 7. Conclusion

Fault injection is a powerful way to attack embedded devices directly at hardware level. Even if the number of faults usually considered for an attack is low, we believe this limit will be overcome soon. Future attacks will consist in a large number of precise fault injections. As seen in Section 3, this provides numerous ways to exploit an existing code at run-time. This NOP-oriented programming has been shown Turing-complete and realistic use-cases based on the STM32 API have been presented. In particular, attacks to dump crypto-key and part of the firmware have been proposed.

In Section 4, an extension of the gem5 simulator that models the NOP-oriented programming has been proposed. Using one of the classic fault models of the literature, realistic use-cases of attacks on cryptographic algorithm and data injection have been presented in Section 5. In these cases, bursts of few hundreds faults have been considered. Such kind of burst allows the attacker to skip large parts of the code, modify registers content and control the execution flow of the program.

Finally, we expose in Section 6 the weakness of software countermeasures as NOP-oriented programming can theoretically bypass them. Hardware countermeasures could be a lead but are rarely taken into account in mainstream systems due to cost and complexity.

Future work will consider building an experimental platform to *i)* reproduce the fault model experienced by Riviere *et al.* and *ii)* extend this model to a NOP-oriented programming, *i.e.*, being able to precisely inject hundred of faults to skip instructions. Moreover, improvements on the simulator can be implemented such as a more complex micro-architecture for the CPU and a more accurate description of other physical effects. Finally, the flexibility of the simulator will be used to investigate and propose countermeasures.

## Acknowledgments

Authors would like to express their acknowledgments to the anonymous reviewers and to Karine Heydemann, Ronan Lashermes and Jean-Louis Lanet for fruitful discussions on early versions of this paper.

## References

[1] Lorena Anghel and Michael Nicolaidis. Cost Reduction and Evaluation of a Temporary Faults-Detecting Technique. In *Design, Automation, and Test in Europe*, pages 423–438. Springer, 2008.

[2] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An In-Depth and Black-Box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2011.

[3] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[4] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures Against Fault Attacks on Software Implemented AES: Effectiveness and Cost. In *5th Workshop on Embedded Systems Security*, 2010.

[5] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 1–6, 2016.

[6] Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. Automated Software Protection for the Masses Against Side-Channel Attacks. *ACM Trans. Archit. Code Optim.*, 15(4), November 2018.

[7] Shivam Bhasin, Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Security Evaluation of Different AES Implementations Against Practical Setup Time Violation Attacks in FPGAs. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 15–21. IEEE, 2009.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[9] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 199–224, 2019.

[10] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, 2008.

[11] Sebanjila K. Bukasa, Ronan Lashermes, Jean-Louis Lanet, and Axel Legay. Let's Shock Our IoT's Heart: ARMv7-M Under (Fault) Attacks. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, pages 33:1–33:6. ACM, 2018.

[12] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy Evaluation of gem5 Simulator System. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7. IEEE, 2012.

[13] Lucian Cojocar, Kostas Papagiannopoulos, and Niek Timmers. Instruction Duplication: Leaky and not Too Fault-Tolerant! In *International Conference on Smart Card Research and Advanced Applications*, pages 160–179. Springer, 2017.

[14] Matthew Cook. Universality in Elementary Cellular Automata. *Complex Systems*, 15, 01 2004.

[15] Matthew Cook. A Concrete View of Rule 110 Computation. *Electronic Proceedings in Theoretical Computer Science*, 1:31–55, Jun 2009.

[16] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS*, 2015.

[17] Chaoqiang Deng and Kedar S Namjoshi. Securing a Compiler Transformation. *Formal Methods in System Design*, 53(2):166–188, 2018.

[18] S. Dolan. mov is Turing-complete. http://stedolan.net/research/mov.pdf.

[19] Emmanuelle Dottax, Christophe Giraud, Matthieu Rivain, and Yannick Sierra. On Second-Order Fault Analysis Resistance for CRT-RSA Implementations. In *IFIP International Workshop on Information Security Theory and Practices*, pages 68–83. Springer, 2009.

[20] Sho Endo, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. A Multiple-Fault Injection Attack by Adaptive Timing Control Under Black-Box Conditions and a Countermeasure. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2014.

[21] Sho Endo, Yang Li, Naofumi Homma, Kazuo Sakiyama, Kazuo Ohta, and Takafumi Aoki. An Efficient Countermeasure Against Fault Sensitivity Analysis Using Configurable Delay Blocks. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 95–102. IEEE, 2012.

[22] Chong Hee Kim and Jean-Jacques Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results and New Countermeasures. In *Proceedings of the 1st IFIP International Conference on Information Security Theory and Practices: Smart Cards, Mobile and Ubiquitous Computing Systems*, WISTP'07, page 215–228. Springer-Verlag, 2007.

[23] Chong Hee Kim and Jean-Jacques Quisquater. New Differential Fault Analysis on AES Key Schedule: Two Faults are Enough. In *International Conference on Smart Card Research and Advanced Applications*, pages 48–60. Springer, 2008.

[24] Fabien Majéric, Eric Bourbao, and Lilian Bossuet. Electromagnetic Security Tests for SoC. In *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2016.

[25] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013.

[26] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal Verification of a Software Countermeasure Against Instruction Skip Attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.

[27] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[28] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Trans. Archit. Code Optim.*, 14(4), December 2017.

[29] Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majeric, and Albert Cohen. A First ISA-Level Characterization of EM Pulse Effects on Superscalar Microarchitectures. In *ARES 2019 - 14th International Conference on Availability, Reliability and Security*, pages 7:1–7:10, Canterbury, United Kingdom, August 2019. ACM Press.

[30] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, 2005.

[31] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015.

[32] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[33] Sergei P Skorobogatov and Ross J Anderson. Optical Fault Induction Attacks. In *International workshop on cryptographic hardware and embedded systems*, pages 2–12. Springer, 2002.

[34] Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA Mechanism on the AES Key Schedule. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 62–74. IEEE, 2007.

[35] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM Using Fault Injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.

[36] Elena Trichina and Roman Korkikyan. Multi Fault Laser Attacks on Protected CRT-RSA. In *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 75–86. IEEE, 2010.

[37] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, 2016.

[38] Loic Zussa, Amine Dehbaoui, Karim Tobich, Jean-Max Dutertre, Philippe Maurine, Ludovic Guillaume-Sage, Jessy Clediere, and Assia Tria. Efficiency of a Glitch Detector Against Electromagnetic Fault Injection. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[39] Loic Zussa, Jean-Max Dutertre, Jessy Clédiere, Bruno Robisson, Assia Tria, et al. Investigation of Timing Constraints Violation as a Fault Injection Means. In *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*, page 11, 2012.